Synchronisation

Synchronisation in distributed systems

The problem with synchronising concurrent activities arises also in non-distributed systems. However, distribution **complicates matters** for many reasons:

- Distributed systems lack a global physical clock
- Distributed systems lack globally shared memory
- Distributed systems suffer from partial failures

In this section, we will study distributed algorithms for:

- Synchronising physical clocks
- Simulating time using logical clocks and preserving event ordering
- Mutual exclusion
- Leader election
- Collecting global state and termination detection
- Distributed transactions
- Detecting distributed deadlocks

Time and distributed systems

Time plays a fundamental role in many applications:

- Execute a predefined action at a given time
- Time stamping objects, data and messages enables the reconstruction of event ordering, which is used in:
    - File versioning
    - Distributed debugging
    - Security algorithms

The problem in distributed systems is to ensure that **all machines "see" the same global time**.

Synchronising physical locks

Computers are not clocks, but **timers**, so in order to guarantee synchronisation:

- A maximum **clock drift rate** ρ is a constant of the timer
- A maximum allowed **clock skew** δ is an engineering parameter
- If two clocks are drifting in opposite directions, during a time interval Δt they accumulate a skew of $2\rho\Delta t$, so a resynchronisation is needed at least every $\frac{\delta}{2\rho}$ seconds

The problem is either:

> **Synchronise all clocks against a single one, usually the one with external and accurate information (accuracy)**

Or:

> **Synchronise all clocks among themselves (agreement)**

Time monotonicity must be preserved in order for things to work. For this, several protocols have been devised.

Positioning and time: GPS

A "side effect" of GPS is that we can get an accurate measurement of time because of how GPS works: position is determined via triangulation from a set of satellites whose position is known. Distance can be measured by the delay of the signal sent and received by the satellite, but for this to work the clock in the satellite and the one in the device **must be in sync**. Since they are not the same physical device, the satellite's time and the device's time are not in sync, so we must take clock skew into account.

Let:

- Δ_r be the unknown deviation of the receiver's clock with regards to the atomic clocks installed on board of satellites
- x_r, y_r, z_r be the unknown coordinates of the receiver
- T_i be the timestamp of a message sent by a satellite i

Suppose that the messages sent by the i-th satellite are received at time T_r according to receiver time, which corresponds to T_{now} in real time. Then:

- $T_{\text{now}} = T_r - \Delta_r$
- $\Delta_i = T_r - T_i$ is the measured delay of the message
- $c \times \Delta_i$ is the measured distance of satellite i

So we can say that:

$$\begin{aligned} c \times \Delta_i &= c \times (T_{\text{now}} - T_i + \Delta_r) \\ &= c \times (T_{\text{now}} - T_i) + c \times \Delta_r \end{aligned}$$

where the first addendum must be equal to the real distance, which is:

$$\sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

With four satellites, we have four equations in four unknowns, including Δ_r, so we can solve them and determine both the receiver position and its clock skew.

We must keep in mind that things are more complex than previous description could suggest:

- Earth is not spherical
- Atomic clocks in the satellites are not perfectly in sync
- The position of satellites is not known precisely
- The receiver's clock has a finite accuracy
- The signal propagation speed is not constant

Regardless, even a cheap GPS receiver can be precise within a range of a few meters and a few tens of nanoseconds.

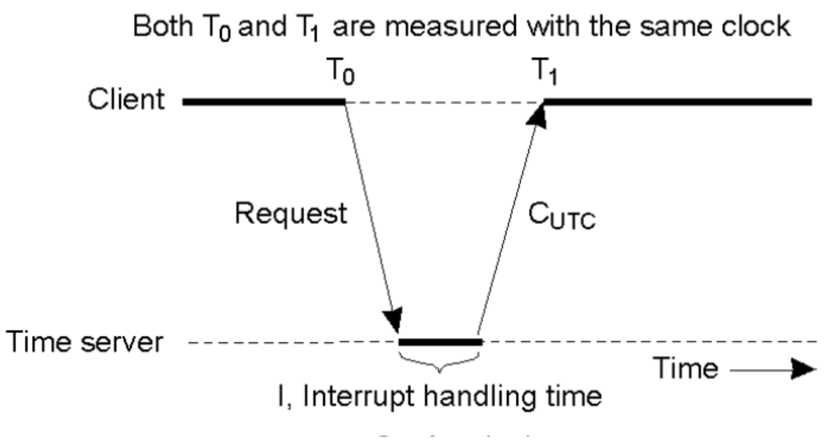
Simple algorithms: Cristian's (1989)

Cristian's algorithm works as follows: periodically, each client sends a request to the time server. Messages are assumed to travel fast with regards to the required time accuracy.

Some problems with this algorithm are:

- Time might run backwards on the client machine (major problem). In order to solve this, change should be introduced gradually (i.e. advance the clock by 9 ms instead of 10 ms at each tick)
- It takes a non-zero amount of time to get the message to the server and back (minor problem): the measured RTT can be used to adjust the time skew

introduced by this problem

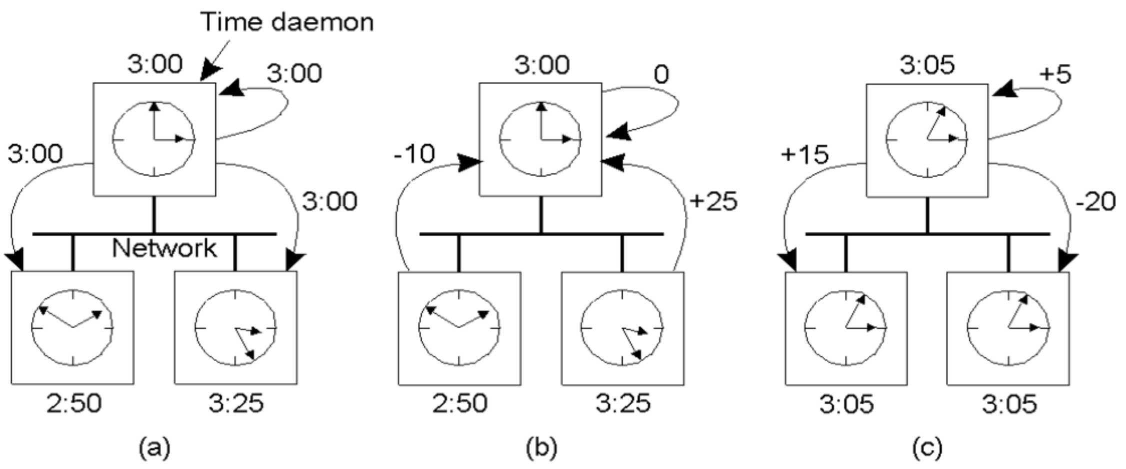

Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

$$T_{round} = T_1 - T_0 - I$$

Simple algorithms: Berkeley (1989)

This algorithm was introduced by Berkeley UNIX.

In this algorithm, the time server is **active**: it collects the time from all clients, averages it and then retransmits the required adjustment.



Network Time Protocol (NTP)

This protocol was designed for UTC synchronisation over large-scale networks and is still used to this day, on top of UDP.

There are about 10 to 20 million NTP clients and servers, which means that this protocol is widely available.

The synchronisation accuracy is:

- About 1 ms over LAN
- About 1 to 50 ms over the Internet

NTP works via hierarchical synchronisation. Each subnet is organised in strata where servers in stratum one are directly connected to a UTC source. Lower strata (i.e. higher levels) provide more accurate information. Leaf servers execute in user workstations.

Connections and strata memberships can change over time.

Synchronisation mechanisms provided by NTP include:

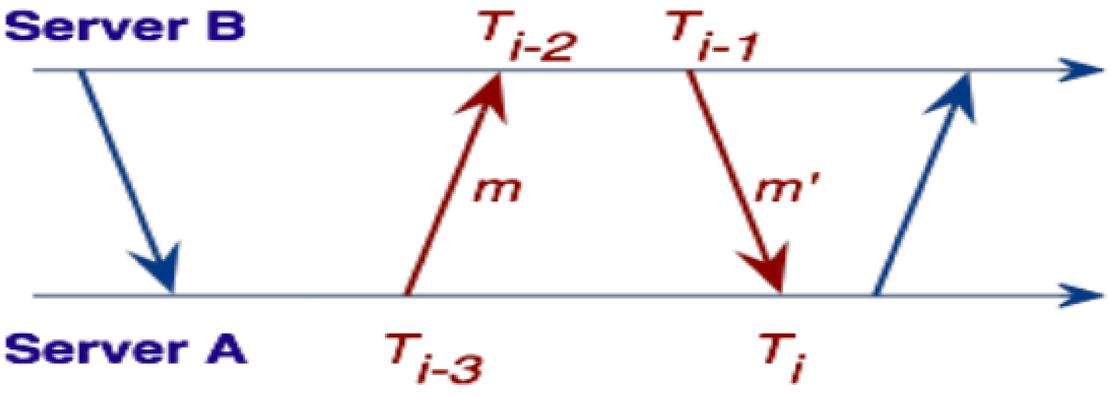
- **Multicast over LAN**: servers periodically multicast their time to other computer on the same network
- **Procedure-call mode**: similar to Cristian's algorithm
- **Symmetric mode**: for higher levels that need the highest level of accuracy

Procedure-call mode and symmetric mode

In these two modes, servers exchange pairs of messages, each bearing timestamps of recent message events: the local time when the previous message between the pairs was sent and received, and the local time when the current message was transmitted.

If t and t' are the messages' transmission times and o is the time offset of the clock at B relative to that at A, then:

$$\begin{aligned} T_{i-2} &= T_{i-3} + t + o \\ T_i &= T_{i-1} + t' - o \end{aligned}$$



This leads to calculate the total transmission time d_i as:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

If we define o_i as:

$$o_i = \frac{T_{i-2} - T_{i-3} + T_{i-1} - T_i}{2}$$

From the first two formulas we have:

$$o = o_i + \frac{t' - t}{2}$$

And since $t, t' \geq 0$, we get:

$$o_i - \frac{d_i}{2} = o_i + \frac{t' - t}{2} - t' \leq o \leq o_i + \frac{t' - t}{2} + t = o_i + \frac{d_i}{2}$$

Thus, o_i is an estimate of the offset and d_i is a measure of the accuracy of this estimate.

Logical time

In many applications, it is sufficient to agree on a time, even if it is not accurate with regards to the absolute time. What matters is the ordering of events, rather than the exact timestamp of them.

Furthermore, if two processes do not interact, there's no need for their clocks to be synchronised.

Scalar clocks

We can define the "happens before" relationship between two event $e \to e'$ as follows:

- If events e and e' occur in the same process and e occurs before e', then $e \to e'$
- If $e = \text{send}(\text{msg})$ and $e' = \text{recv}(\text{msg})$, then $e \to e'$

If neither $e \to e'$ nor $e' \to e$ are true, we say that the two events e, e' are **concurrent** and we can write $e || e'$.

The "happens before" relationship captures **potential causal ordering** among events: two events can be related by the "happens before" relationship **even if there is no causal connection** between them. Also, since information can flow in ways other than message passing, two events may be causally related **even if neither of them happens before** the other.

Leslie Lamport invented a simple mechanism by which the "happened before" ordering can be captured numerically, using integers to represent the clock value:

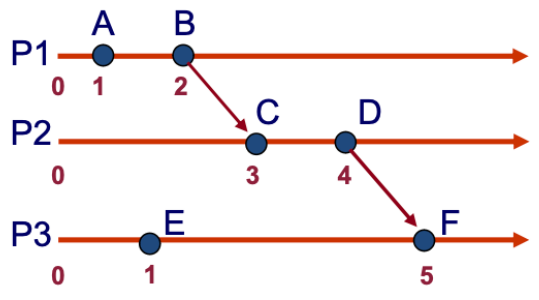
- Each process p_i keeps a logical scalar clock L_i, which starts at zero and is incremented before the process sends a message
- Each message sent by p_i is timestamped with L_i
- Upon receipt of a message, p_i sets L_i to $\max(\text{msg timestamp}, L_i) + 1$

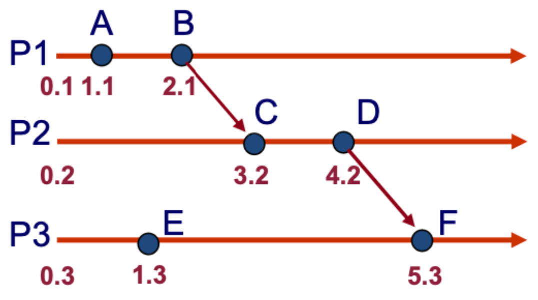
It can be easily shown via induction on the length of any sequence of events relating two events e, e' that:

$$e \to e' \implies L(e) < L(e')$$

Note

Only partial ordering is achieved in this way, although total ordering can be obtained trivially by attaching process IDs to clocks.





Totally ordered multicast

One implementation of scalar clocks is used to achieve **totally ordered multicast**.

Let's look at an example first: we want to manage bank accounts in a distributed manner. One customer deposits $100 in his account and the bank needs to add a 1% interest rate to that customer's account.

Since in this system, updates are propagated to all locations, if updates are not done in the same order, the results between locations could be inconsistent.

This is solved through totally ordered multicast:

- Messages are sent and acknowledged via multicast
- All messages, including acks, carry a timestamp with the sender's scalar clock
- Receivers (including the sender) store all messages in a queue, ordered according to their timestamp
- Eventually, all processes in the system have the same messages in their queues
- A message is delivered to the application **only when it is at the highest in the queue** and **all its acks have been received**

Vector clocks

As we said earlier, using scalar clocks, the following holds:

$$e \rightarrow e' \implies L(e) < L(e')$$

The opposite, however, does not, i.e. if $e||e'$. A solution to this problem is offered by **vector clocks**.

In vector clocks, each process p_i maintains a vector V_i of N values, where N is the number of processes, such that:

- $V_i[i]$ is the number of events that have occurred at P_i
- If $V_i[j] = k$, then P_i knows that k events have occurred at P_j

The rules to update the vector clock are as follows:

- Initially, $V_i[j] = 0$ for all i, j
- A local event in P_i causes an increment of $V_i[i]$
- P_i attaches a timestamp $t = V_i$ in all messages it sends, incrementing $V_i[i]$ just before sending the message
- When P_i receives a message containing the timestamp t, it sets $V_i[j] = \max(V_i[j], t[j])$ for all $j \neq i$, and then increments $V_i[i]$

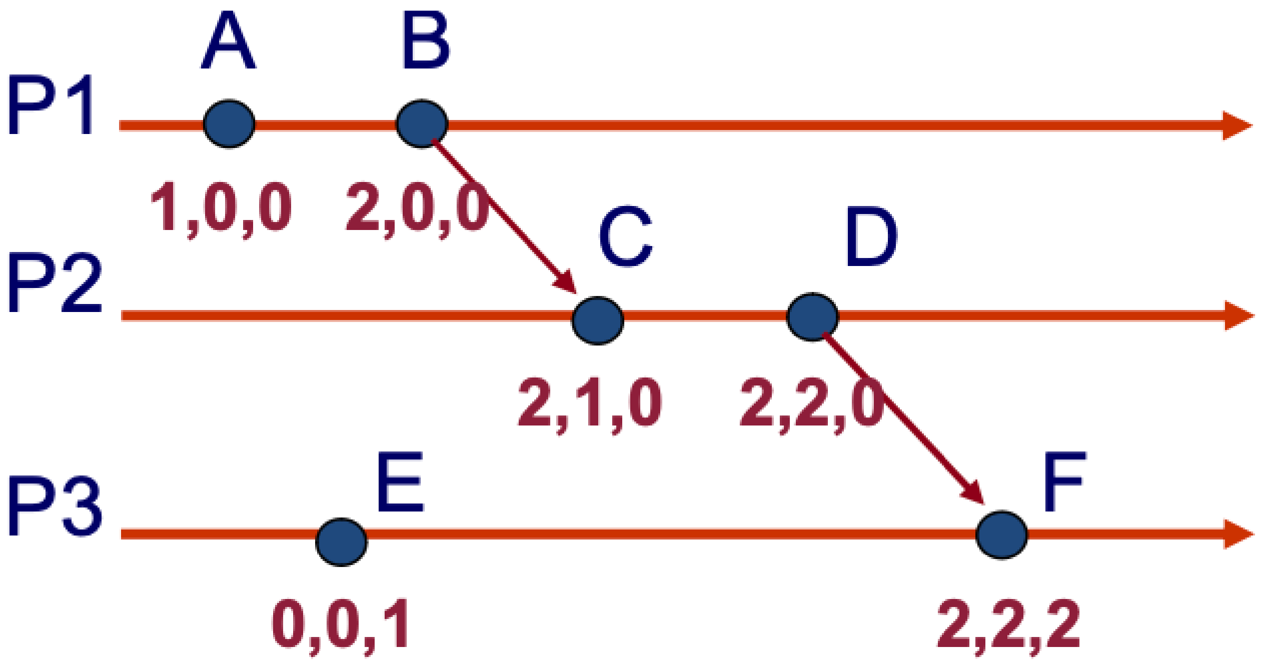
We can define some partial ordering definitions now:

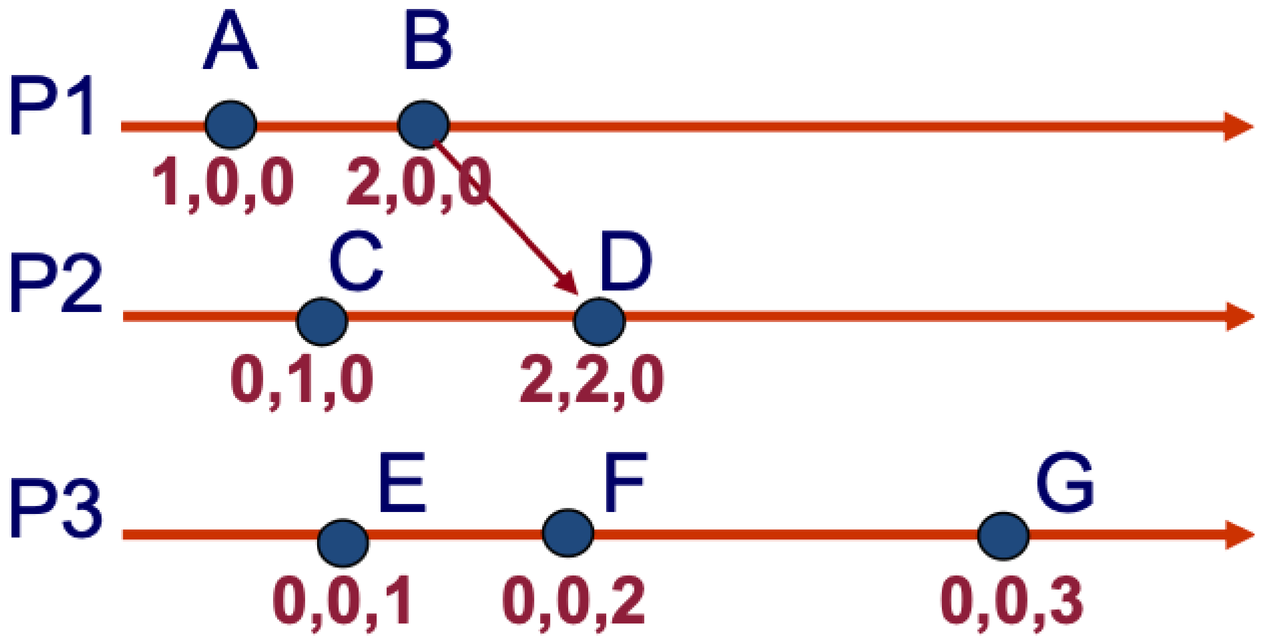
$$\begin{aligned}
V = V' &\iff V[j] = V'[j], \forall j \\
V \leq V' &\iff V[j] \leq V'[j], \forall j \\
V < V' &\iff V \leq V' \land V \neq V' \\
V \mid\mid V' &\iff \neg(V < V') \land \neg(V' < V)
\end{aligned}$$

We can then determine causality between events like so:

$$\begin{aligned}
e \rightarrow e' &\iff V(e) < V(e') \\
e || e' &\iff V(e) \mid\mid V(e')
\end{aligned}$$

Now, by looking at timestamps **only**, we are able to determine whether two events are causally related or concurrent:





Vector clocks for causal delivery

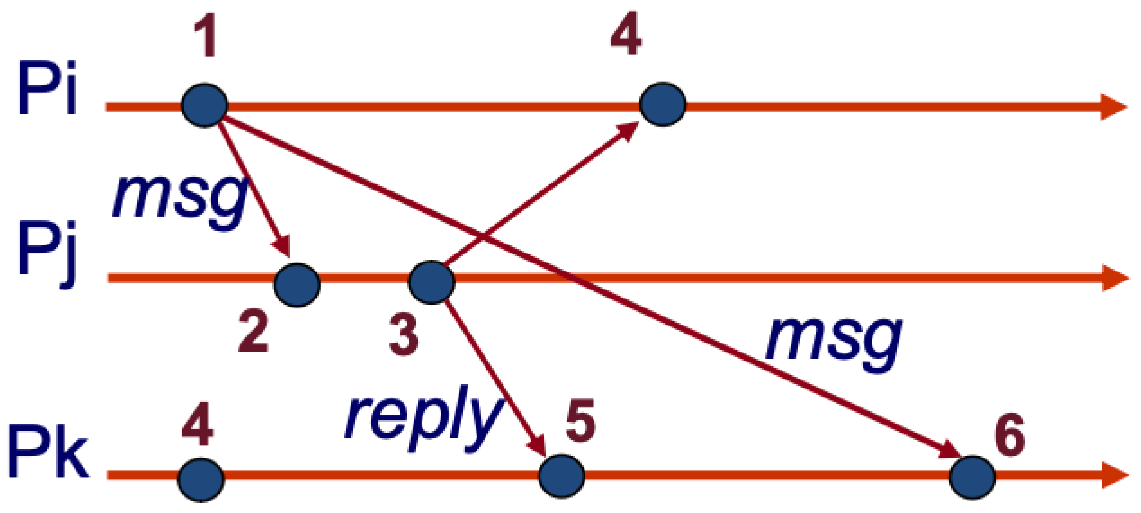
A slight variation of vector clocks can be used to implement causal delivery of messages in a totally distributed way.

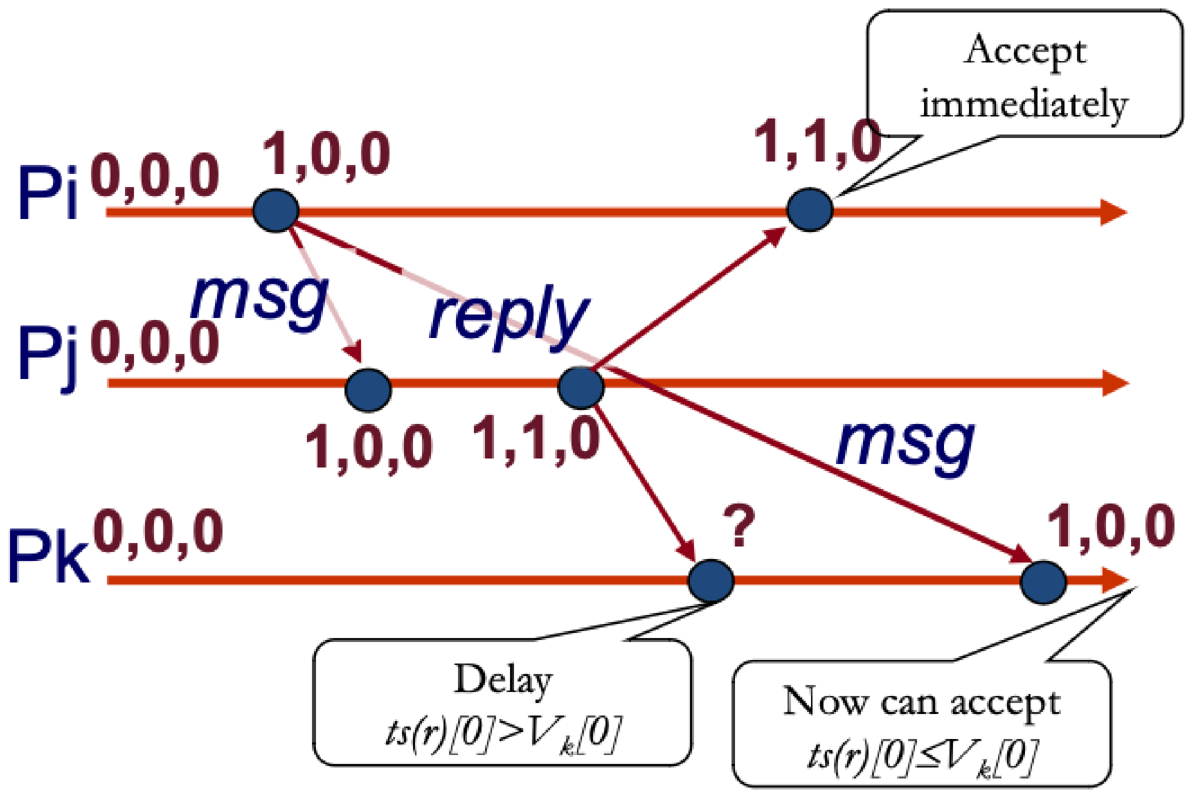
One application of this is bulletin boards:

- Messages and replies are sent to all boards, in parallel
- The only thing that's needed is to preserve the ordering between messages and

replies
- Totally ordered multicast is too strong: if `M1` arrives before `M2`, it does not necessarily mean that the two are related

Using vector clocks, we could increment the index of a specific process only when messages are sent and not when they are received. Furthermore, a reply is held until the previous message is received as well:





Mutual exclusion

Mutual exclusion is required to prevent interference and ensure consistency of resource access. This is a critical section problem, typical of operating systems. However, in distributed systems, there is no shared memory.

Mutual exclusion requires that:

- **Safety property**: at most one process may execute in the critical section at a time
- **Liveness property**: all requests to enter or exit the critical section eventually succeed (this means there should be no deadlocks and no starvation)
- Optional: if one request happened before another one, then entry is granted in that order

> **Note**
>
> In this section, we assume to have **reliable channels and processes**.

The simplest solution to this problem is to have a server act as a coordinator, choosing which processes gain access to that resource and in what order. This emulates a centralised solution.

The coordinator manages the resource's lock through a token and resource access requests are obtained with messages to the coordinator.

This method makes it fairly easy to guarantee mutual exclusion and fairness, but the fact that a single server acts as the coordinator brings performance bottlenecks and a single point of failure to the whole system.

Mutual exclusion with scalar clocks

We can use the notion of scalar clocks to implement mutual exclusion: to request access to a resource, a process P_i multicasts a resource request message m, with a timestamp T_m, to all processes (including itself).

Upon receipt of m, a different process P_j acts in different ways depending on whether it is holding the resource at that moment or not:

- If the process P_j **does not hold** the resource P_i is requesting and is **not interested in holding the resource** "in the near future", then P_j sends an

acknowledgement to P_i
- If the process P_j **is holding** the resource, then it puts the request into a local queue ordered according to T_m (in this case, process IDs are used to break ties)
- If the process P_j **is interested in holding** the resource and has already sent out a request, P_j compares the timestamp T_m with the timestamp of its own request. If T_m is the lowest one, P_j sends an ack to P_i; otherwise, it puts the request into the local queue

On releasing a resource, a process acknowledges all the requests queued while using it.

The access to a resource is granted to a process when its request has been acknowledged by **all the other processes**.

A token ring solution

With this solution, processes are logically arranged in a ring, regardless of their physical connectivity. Access is granted by a token that is forwarded along a given direction on the ring:

- A process that is not interested in accessing the resource simply forwards the token
- A process is granted access to a resource if it retains the token
- A process releases a resource if it forwards the token

Leader election

Many distributed algorithms require a process to act as a **coordinator** (or some other special role). Here, the problem is having **everyone agree on a new leader**.

In order to solve this problem, we have to make the assumption that **nodes are distinguishable**, otherwise there would be no way to perform a selection.

We also assume that we are working in a closed system: that is, processes know each other and their IDs.

At the end of an election, the non-crashed process with the highest ID must be the winner and every other non-crashed process must agree on this.

Different algorithms differ on the selection process.

The bully election algorithm

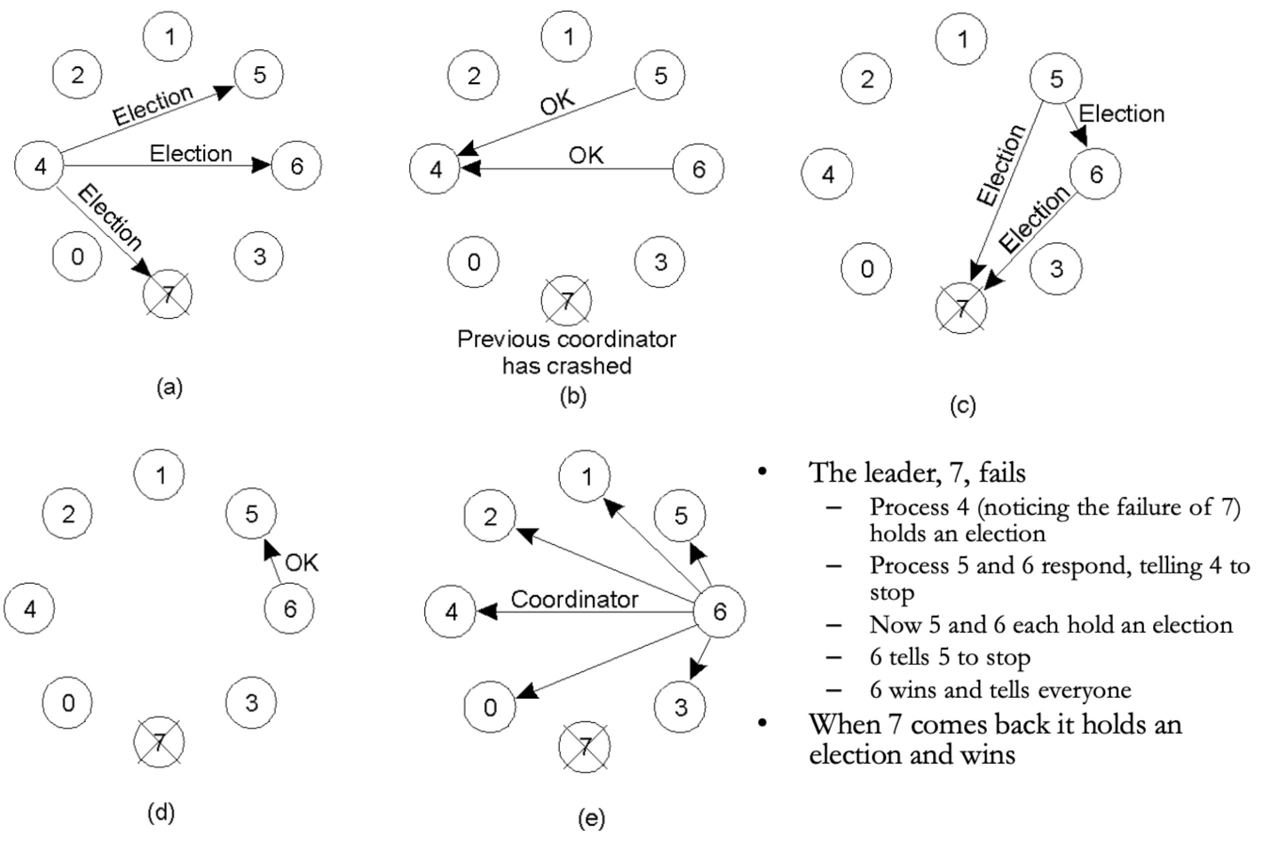
In order to have the bully election algorithm work, we must make some further assumptions:

- The system has reliable links
- It is possible to decide who has crashed (requires a synchronous system)

The algorithm works as follows:

1. When any process notices that the current coordinator is no longer responding, it initiates an **election**: the process sends an `ELECT` message, including its ID, to all other processes with higher IDs
2. If no one responds, **the process who sent the message wins** and sends a `COORD` message to the processes with lower IDs
3. If a different process receives the `ELECT` message and has a higher ID than the sender, it responds, stopping the sender from being a candidate
4. If a process that was previously down starts to operate again, it holds an election. If this process happens to be the highest-numbered process currently running, it automatically wins the election and takes over the coordinator's job

Below is an example of this algorithm:



- The leader, 7, fails
  - Process 4 (noticing the failure of 7) holds an election
  - Process 5 and 6 respond, telling 4 to stop
  - Now 5 and 6 each hold an election
  - 6 tells 5 to stop
  - 6 wins and tells everyone
- When 7 comes back it holds an election and wins

A ring-based algorithm

Let's now assume that among the nodes in a system there's a ring topology, either physical or logical.

When a process detects a leader failure, it sends an `ELECT` message, containing its ID, to the closest alive neighbour.

Upon receipt of the election message, a process P can do the following two things:

- If P is not in the list inside the message, it adds P and propagates the message to the next neighbour
- If P is in the list inside the message, the message type is changed to `COORD` and recirculated in the system

On receiving a `COORD` message, a node considers the process with the highest ID the new leader and is also informed about the remaining members of the ring.

In this topology, multiple messages may circulate at the same time; they will eventually converge to the same content.

Capturing global state

The global state of a distributed system consists of the local state of each process, together with the message in transit over the links.

The global state of a distributed system is useful to know for debugging, termination detection, deadlock detection and more.

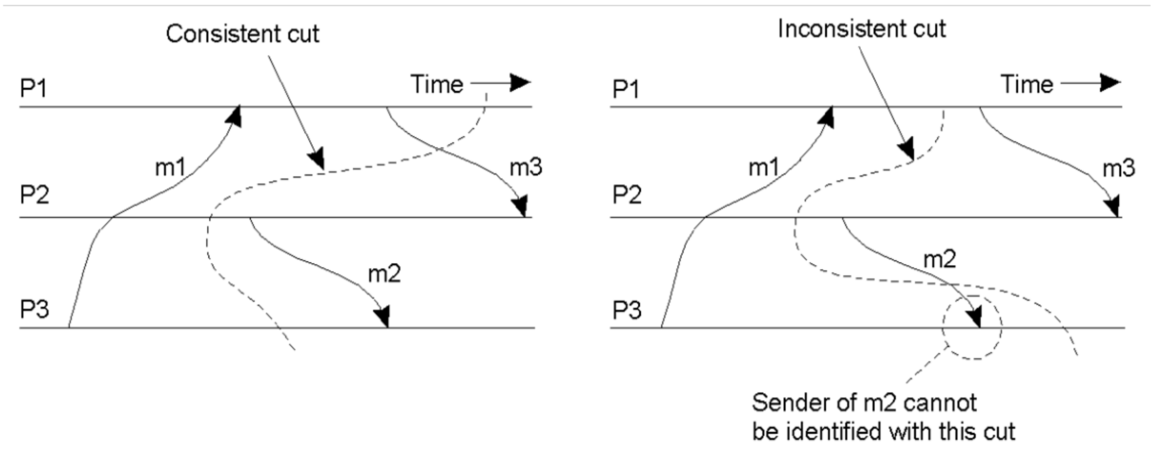
Capturing the global state of a distributed system would be trivial if we could access a global clock but alas, we cannot. We must accept recording the state of each process at, potentially, different times.

Cuts and distributed snapshots

A distributed snapshot reflects a consistent global state in which the distributed system might have been.

Particular care must be taken when reconstructing the global state to preserve consistency: if a message receipt is recorded, the message sending must be recorded as well. The contrary, however, is not required.

The conceptual tool we will be using in order to preserve consistent global states is the **cut**:



Formally, a cut of a system S composed of N processes can be defined as the union of the histories of all its processes up to a certain event:

$$C = h_1^{k_1} \cup h_2^{k_2} \cup \ldots \cup h_N^{k_N},\ h_i^{k_i} = \{e_i^0, e_i^1, \ldots, e_i^{k_i}\}$$

A cut C is said to be **consistent** if, for any event e it includes, it also includes all the events that happened before e. Formally:

$$\forall e, f : e \in C \land f \to e \implies f \in C$$

The Chandy-Lamport algorithm (1985)

Let's take a look at the Chandy-Lamport snapshot algorithm.

In order to explain this algorithm, we must make some assumptions:

- The system is a strongly connected graph
- Messages use FIFO queues
- Nodes are reliable
- Links are reliable

Let us now explain the algorithm:

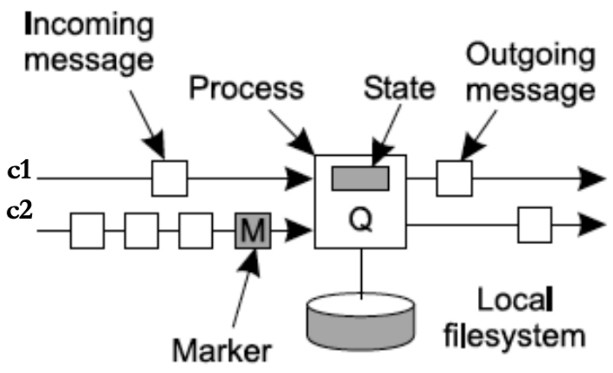
- Any process p may initiate a snapshot by:

    - Recording its internal state
    - Sending a token on all outgoing channels
    - Start recording a local snapshot
- Upon receiving a token, a process q acts differently depending on whether it was already recording a snapshot or not:
    - If the process was not recording a snapshot at the time of the receipt of the token, it follows the same procedure as p
    - Regardless of the state of the snapshot, q stops recording the incoming messages on the channel the token arrived on

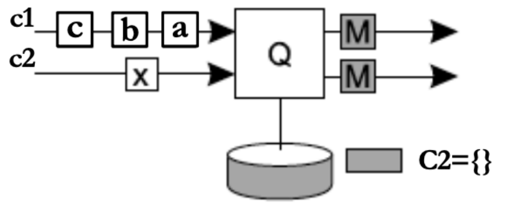
Recording messages in this algorithm works like this:

- If a message arrives on a channel which is recording messages, the arrival of the message is recorded; after that, the message is processed normally
- If a message arrives on a channel that is not recording messages, the message is only processed, but not recorded

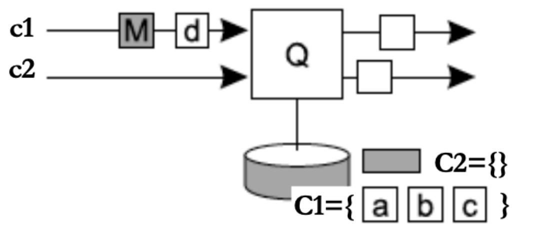
Each process considers the snapshot as finished when tokens have arrived on all its incoming channels. Afterwards, the collected data can be sent to a single collector of the global state.



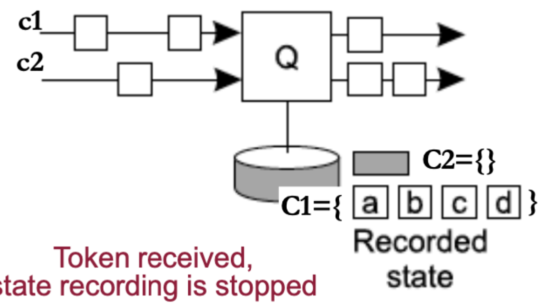
Normal processing, first marker about to be received



Token just received for the first time, state saved, token forwarded to outgoing links, begin recording messages



Recording of messages from the incoming links from which a token has not been received, yet



Token received, state recording is stopped

Characterising the observed state

Theorem

The distributed snapshot algorithm selects a consistent cut.

The proof to the theorem is provided below.

Proof

Let e_i and e_j be two events occurring at p_i and p_j respectively, such that $e_i \rightarrow e_j$. Suppose e_j is part of the cut and e_i is not. This means e_j occurred before p_j saved its state, while e_i occurred after p_i saved its state. If $p_i = p_j$, this is trivially impossible, so suppose $p_i \neq p_j$. Let $m_1, \dots, m_h$ be the sequence of messages that give rise to the relation $e_i \rightarrow e_j$. If e_i occurred after p_i saved its state, then p_i sent a marker ahead of $m_1, \dots, m_h$. By FIFO ordering over channels and by the marker propagating rules, it results that p_j received a marker ahead of $m_1, \dots, m_h$. By the marker processing rule, it results that p_j saved its state before receiving $m_1, \dots, m_h$, i.e. before e_j, which contradicts our initial assumption$\blacksquare$

Some observations

Important

The distributed snapshot algorithm does not require blocking of the computation. Collecting the snapshot is interleaved with processing.

What happens if the snapshot is started at more than one location at the same time?

✓ **This problem is easily dealt with by associating an identifier to each snapshot, set by the initiator**

Termination detection

If we want to know when a computation has completed or deadlocked, we can use a distributed snapshot, granted that the channels were empty when the snapshot was recorded.

A simple solution to this problem comes from Tanenbaum:

- Let's call the **predecessor** of a process p the process q from which it got the first marker. The successors of p are all those processes p sent the marker to.
- When a process p finishes its part of the snapshot, it sends a `DONE` message back to its predecessor only if two conditions are met:
    1. All of p's successors have returned a `DONE` message
    2. p has not received any message between the point it recorded its state and the point it had received the marker along each of its incoming channels
- In any other case, p sends a `CONTINUE` message
- If the initiator receives all `DONE` messages, the computation is over; otherwise, another snapshot must be created

Diffusing computations: Dijkstra-Scholten termination detection

In a diffusing computation, initially all processes are idle, except for the `init` process.

Furthermore, a process is activated only by sending it a message.

In this case, the termination condition is still the same as before: when the processing is complete at each node and there are no more messages in the system, all processes are idle.

For diffusing computations, the Dijkstra-Scholten termination detection algorithm is used. Its key concepts are:

- Create a tree out of the active processes
- When a node finishes processing and is a leaf, it can be pruned from the tree
- When only the root remains and it has completed processing, the system has terminated

Some challenges to solve this problem are:

- How to create a tree and always keep it acyclic
- How to detect when a node becomes a leaf

The Dijkstra-Scholten algorithm works as follows:

- Each node keeps track of nodes it sends a message to: its children
- If a node was already awake when the message arrived, then it is already part of the tree and should not be added as a child of the sender
- When a node has no more children and it is idle, it tells its parent to remove it as a child

Comparison of termination detection approaches

The following table aims to compare the different approaches we've looked at for termination detection:

Distributed snapshots	Dijkstra-Scholten
Overhead is one message per link	Overhead depends on the number of messages in the system
Result collection is costly	Does not involve processes that never activate
If the system does not terminate, the algorithm needs to be run again	A termination is detected only when the last ack is received

---

Distributed transactions

In order to protect a shared resource against simultaneous access by several concurrent processes, we can use **transactions**: sequences of operations that are defined with appropriate programming primitives and have ACID properties.

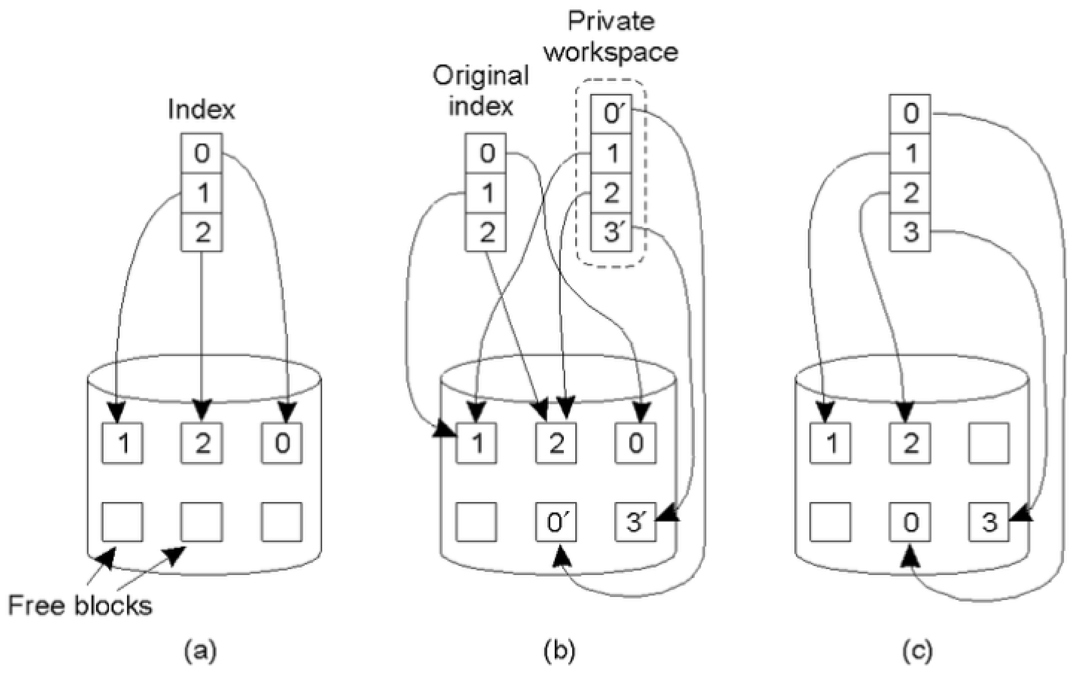
There are several transaction types:

- **Flat** transactions: they are ACID transactions, as defined above
- **Nested** transactions: they are constructed from **sub-transactions**
  - Sub-transactions can be undone once committed if their parent transaction aborts: the durability property only applies to top-level transactions
  - Sub-transactions conceptually operate on a private copy of the data: if the sub-transaction aborts, the private copy disappears; if it commits, the modified private copy is available to the next sub-transaction
  - Typically, each sub-transaction runs on a different host
- **Distributed** transactions: they account for data distribution and are essentially flat transactions on distributed data. In order to have ACID properties, they need **distributed locking**

Achieving atomicity

In order to achieve atomicity in distributed transactions, we can apply several approaches:

1. **Private workspace**: copy what the transaction modifies into a separate memory space, creating **shadow blocks** of the original files. If the transaction is aborted, this private workspace is deleted; otherwise it is copied into the parent's workspace. This method can be optimised by replicating the index and not the whole file and works fine for the local part of distributed transactions



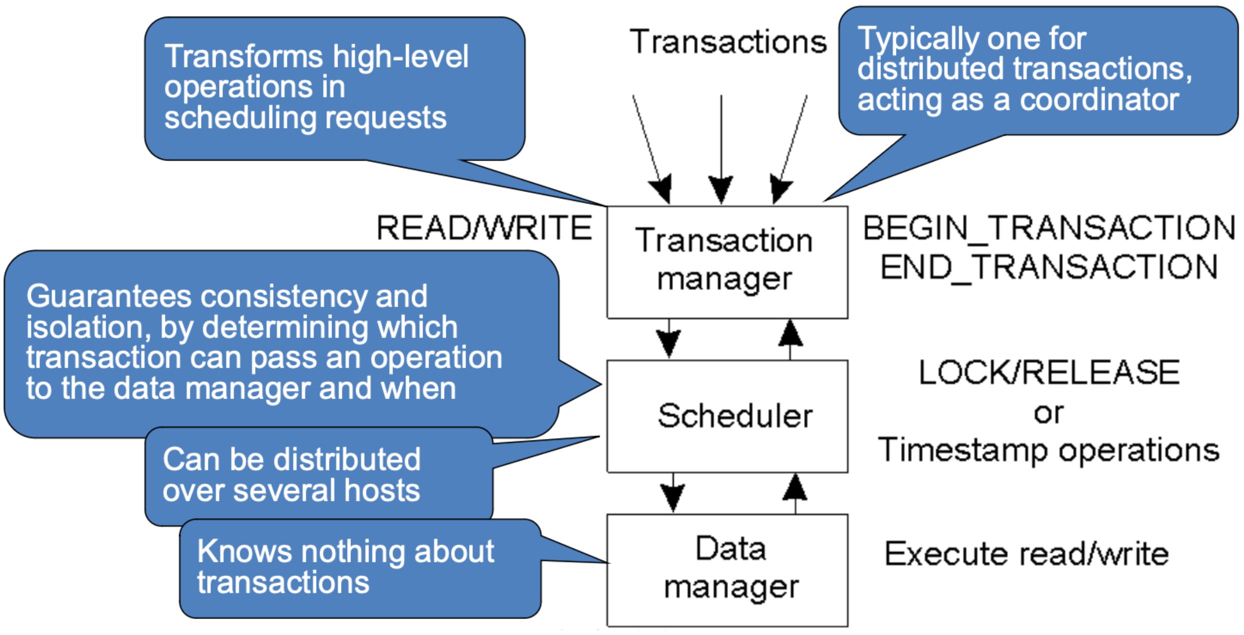
2. **Writeahead log**: files are modified in place (so commit is much faster), but a log is kept with the transaction that made changes, which file and/or block was modified and the old and new values. After the log is written successfully, the file is actually modified. If the transaction succeeds, the commit is written to the log; if it aborts, the original state of the file is restored based on the same logs, starting at the end (*rollback* operation)

Controlling concurrency

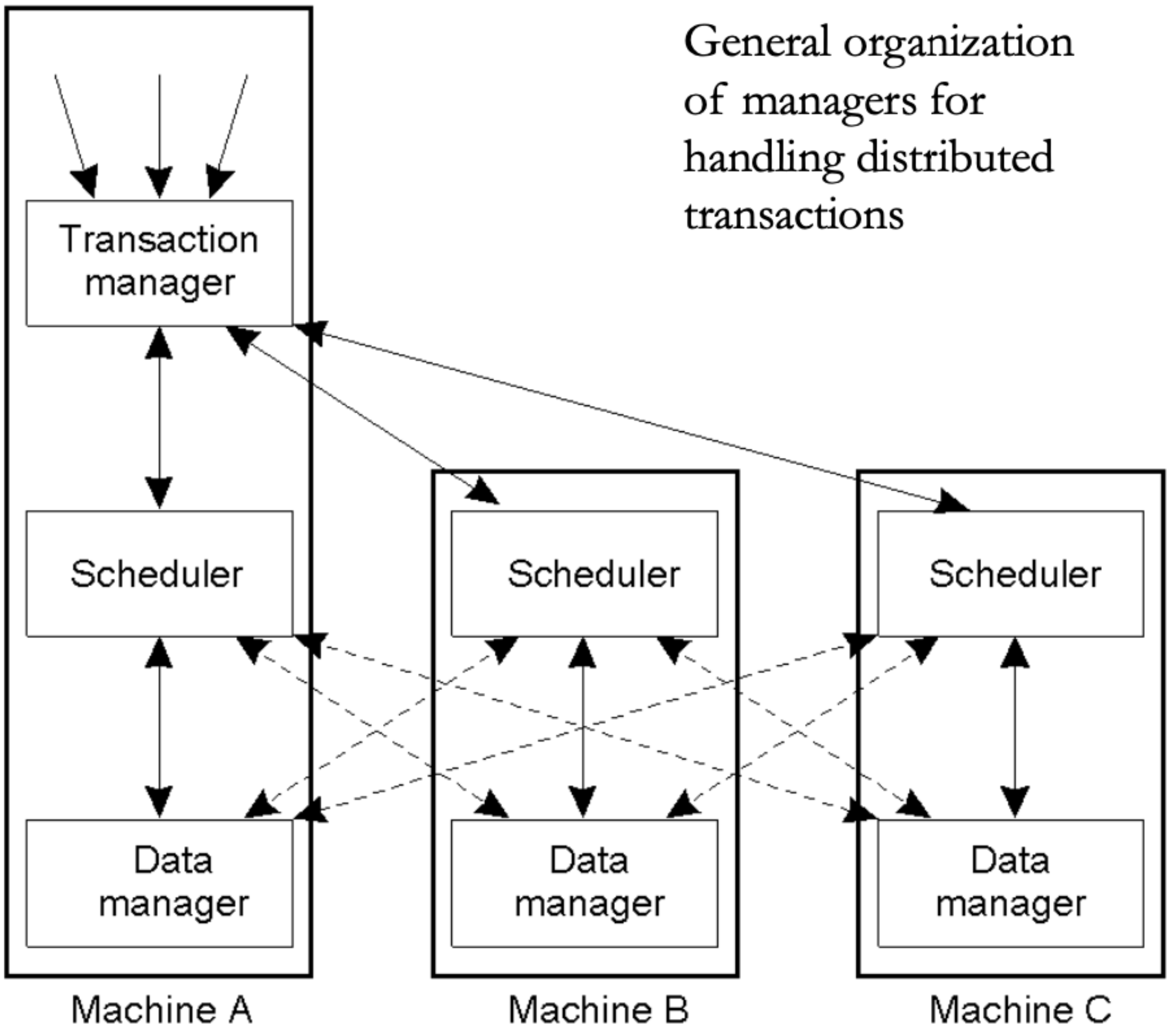
In order to control concurrency, we need some components:

- A **transaction manager**
- A **scheduler**
- A **data manager**

Their duties are described in the picture below:


Transforms high-level operations in scheduling requests
Transactions
Typically one for distributed transactions, acting as a coordinator
READ/WRITE
Transaction manager
BEGIN_TRANSACTION END_TRANSACTION
Guarantees consistency and isolation, by determining which transaction can pass an operation to the data manager and when
Scheduler
LOCK/RELEASE or Timestamp operations
Can be distributed over several hosts
Knows nothing about transactions
Data manager
Execute read/write

To handle distributed transactions, the scheduler is usually distributed as well, while the transaction manager is centralised:



General organization of managers for handling distributed transactions

Serialisability

Several interleavings are possible for the same transaction, but only the ones that correspond to some linearisation of the involved transactions are legal.

Transaction systems must ensure that operations are interleaved correctly, but should also free the programmer from the burden of programming mutual exclusion: if two operations in a schedule are in conflict with each other, the scheduler should fix that.

Two-Phase Locking (2PL)

A very secure system to ensure no conflicts can arise between operations is set by Two-Phase Locking (2PL):

- The scheduler tests whether the requested operation conflicts with another one that has already received the lock. If so, the operation is delayed
- Once a lock for transaction T has been released, T can no longer acquire it

A variant of 2PL called **Strict 2PL** can also be used. In this variant, all locks are released at the same time, preventing cascaded aborts by requiring the shrink phase to take place only after transaction termination.

It is proven that 2PL leads to serialisability, but it may very well deadlock several transactions.

2PL can be implemented in many different ways:

- **Centralised** 2PL: the transaction manager contacts a centralised lock manager, receives lock grants and interacts directly with the data manager; then returns the lock to the lock manager
- **Primary** 2PL: multiple lock managers exists and each data item has a primary copy on a host; the lock manager on that host is responsible for granting locks. The transaction manager is responsible for interacting with the data managers
- **Distributed** 2PL: it assumes that data may be replicated on multiple hosts, so the lock manager on a host is responsible for granting locks on the local replica and for contacting the data manager

Pessimistic timestamp ordering

In distributed transactional systems, the system assigns a timestamp to each transaction using logical clocks.

Write operations on data item x are recorded in tentative versions, each with its own write timestamp $\mathrm{ts}_{\mathrm{wrt}}(x_i)$ until a commit is performed. We'll refer to the write timestamp of the committed version of x as $\mathrm{ts}_{\mathrm{wrt}}(x)$.

Each data item x also has a read timestamp $\mathrm{ts}_{\mathrm{rd}}(x)$, which specifies the latest time at which a transaction read from x.

In pessimistic timestamp ordering, the scheduler operates as follows:

- If the scheduler receives a **write** operation from transaction T at time ts:
    - If $\mathrm{ts} > \mathrm{ts}_{\mathrm{rd}}(x)$ and $\mathrm{ts} > \mathrm{ts}_{\mathrm{wrt}}(x)$, it performs a tentative write on x_i with timestamp $\mathrm{ts}_{\mathrm{wrt}}(x_i)$
- If the scheduler receives a **read** operation from transaction T at time ts:
    - If $\mathrm{ts} > \mathrm{ts}_{\mathrm{wrt}}(x)$, let x_{sel} be the latest version of x with the write timestamp lower than ts. If x_{sel} is committed, perform a read on x_{sel} and set $\mathrm{ts}_{\mathrm{rd}}(x) = \max(\mathrm{ts}, \mathrm{ts}_{\mathrm{rd}}(x))$; otherwise, wait until the transaction that wrote version x_{sel} commits or abort, then reapply the rule
    - Otherwise abort transaction T, since the read request arrived too late

This algorithm is deadlock-free, but a lot of transactions will be aborted because they arrive too late.

Optimistic timestamp ordering

This timestamp ordering technique is based on the assumption that conflicts are a rare occurrence. Therefore, transactions should commit without caring about the other transactions. Possible conflicts should be fixed at a later time.

This algorithm works as follows:

- Timestamp data items with the start time of the transaction
- When the transaction commits, if any items have been changed since the start of it, the transaction is aborted; otherwise it is committed normally

This timestamp ordering is best implemented with private workspaces. It is deadlock-free and allows for maximum parallelism. However, under heavy load, there may be too many rollbacks.

Distributed deadlocks

Distributed deadlocks are similar to the ones that we encounter on local systems, but worse to deal with since, in distributed systems, resources are spread across multiple machines.

In order to solve distributed deadlocks, multiple approaches are used:

- **Ignore the problem**
- Deadlock detection: generally based on killing one of the processes that are causing the deadlock
- Deadlock prevention
- **Deadlock avoidance**: this approach is not used in distributed systems, since it implies a priori knowledge about resource usage

In distributed deadlocks, distributed transactions are helpful, since aborting a single transaction is much less disruptive than killing an entire process.

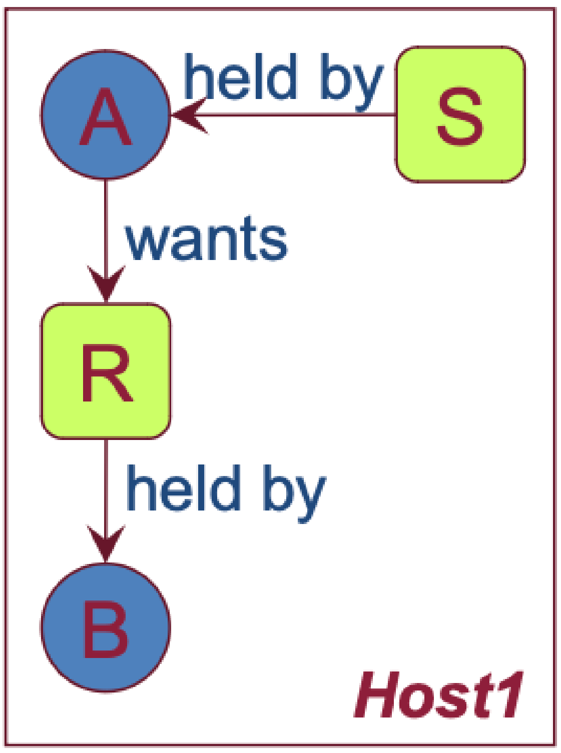
Centralised deadlock detection

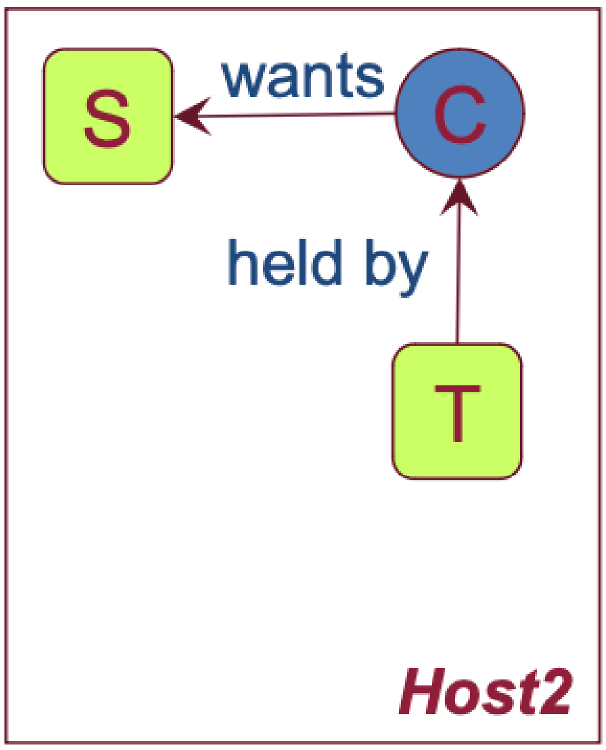
For centralised deadlock detection, each machine contains a resource graph for its own resources and reports it to a coordinator.

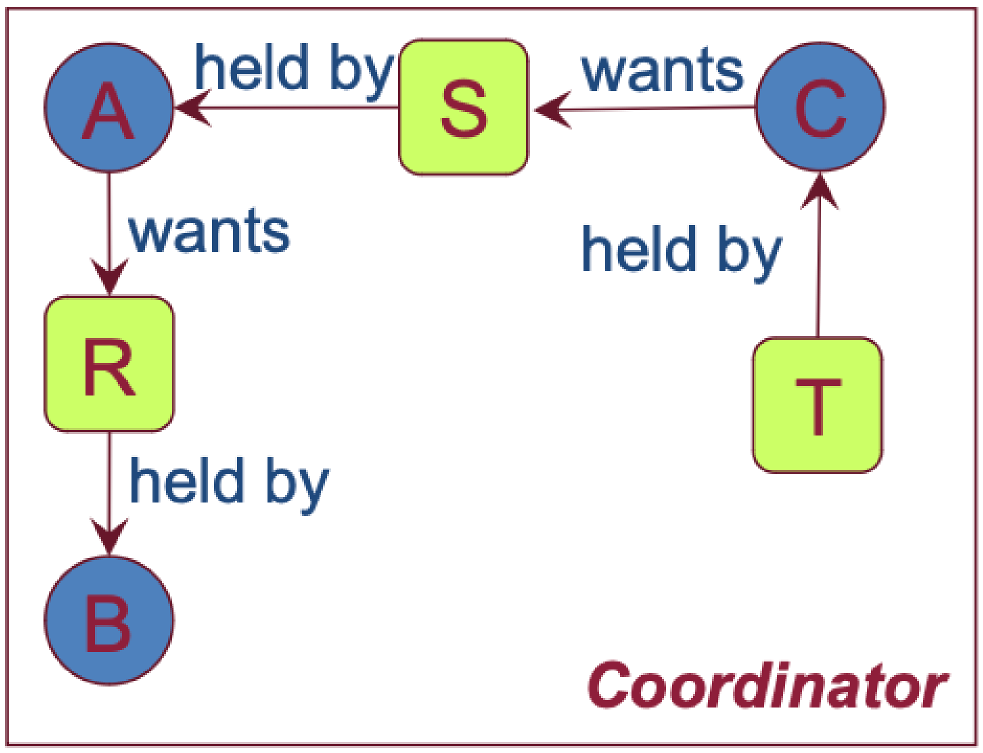
There are different ways for the coordinator to gather this information:

- Whenever an arc is added or deleted, a message is sent to the coordinator with the update
- Periodically, every process sends a list of arcs added or deleted since the last update
- The coordinator can request information on-demand

Unfortunately, none of these ways work well because of possible false deadlocks. Let's look at the image below for an example of this:







If B releases R and acquires T and the coordinator receives data from host 2 before receiving data from host 1, a false deadlock occurs.

Distributed deadlock detection

In distributed deadlock detection, there is **no coordinator** in charge of building the global wait-for graph.

Processes are allowed to request multiple resources simultaneously. When a process gets blocked, it sends a probe message to the process holding resources it wants to acquire. A **cycle is detected** if the **probe makes it back to its initiator**.

If a deadlock is detected, there are several ways to recover:

- **The initiator commits suicide**: this recovery method can bring to unnecessary aborts if more than one initiator detects a loop
- **The initiator picks the process with highest ID and kills it**, although this requires each process to add its identifier to the probe message

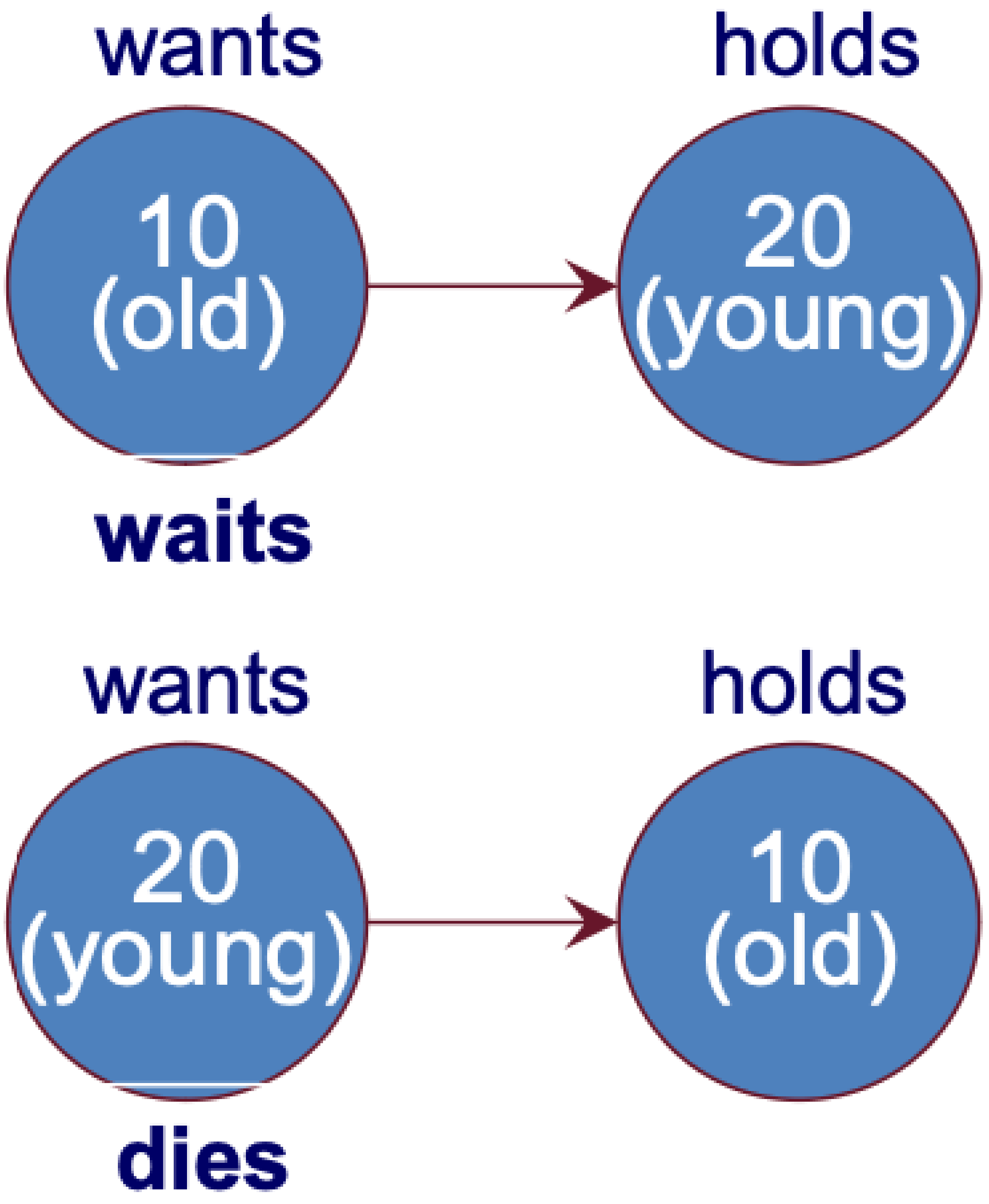
In practice, 90% of all deadlock cycles involve just two processes (in databases at least).

Distributed deadlock prevention

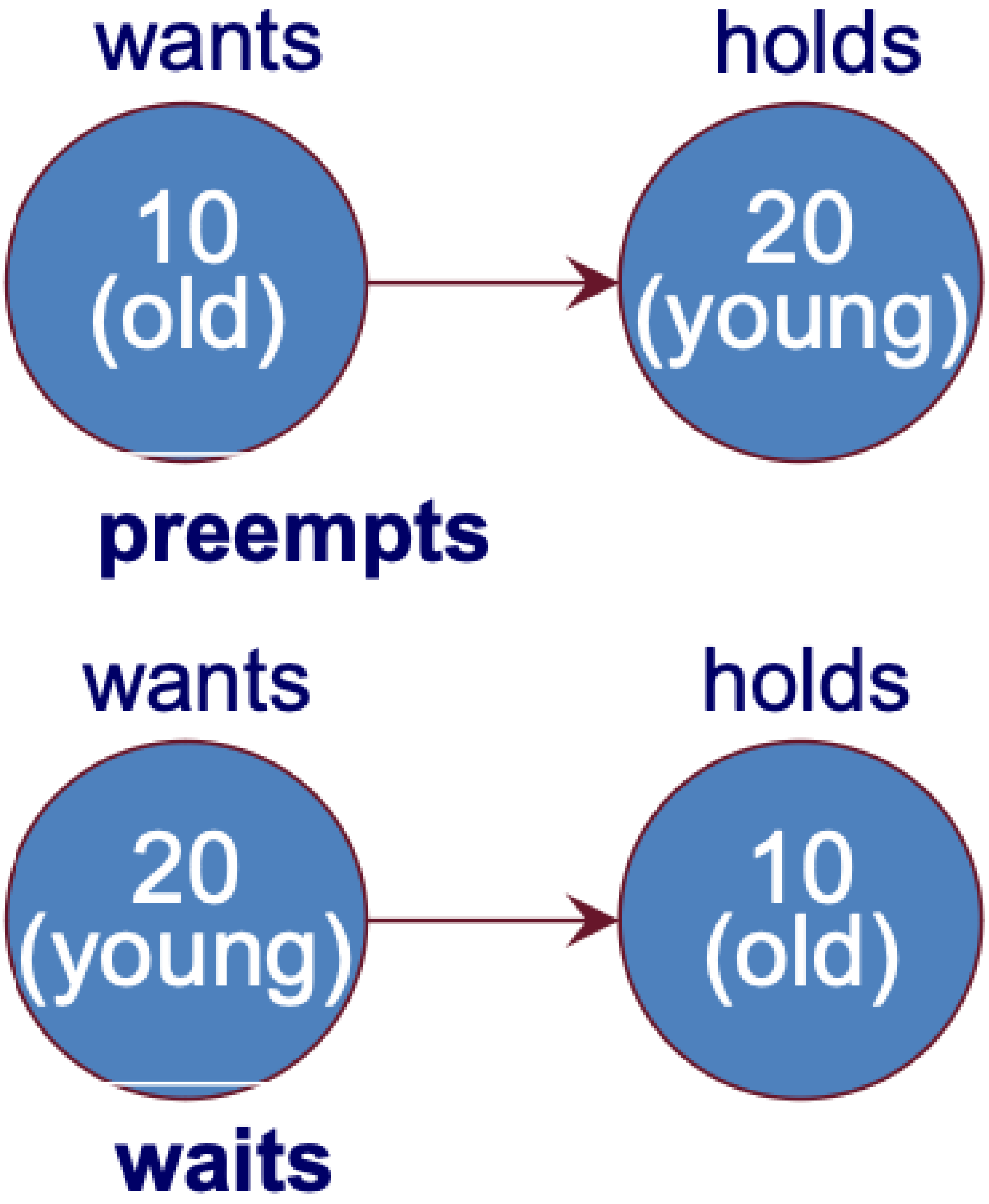
Deadlock prevention, differently from deadlock detection, makes deadlocks **impossible by design**.

The **wait-die** algorithm, for instance, uses global timestamps to avoid deadlocks: when a process A is about to block for a resource that another process B is using, allow A to wait only if A has a lower timestamp (that is, it is **older**) than B; otherwise, kill process A.

Following a chain of waiting processes, the timestamps will always increase, so no cycles are possible and therefore, no deadlocks are either.



If a process can be preempted instead (i.e. its resource can be taken away), an alternative can be devised: the **wound-wait** algorithm: preempting the young process aborts its transaction. The young process will then wait to reacquire the resource again.



In the wait-die algorithm, young processes may die many times before the old ones release the resources; in wound-wait, this is not the case.